(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number : **0 317 080 B1**

(12) **EUROPEAN PATENT SPECIFICATION**

(45) Date of publication of patent specification : **14.06.95 Bulletin 95/24**

(51) Int. Cl.^{6} : **G06F 11/00,** G06F 9/44

(21) Application number : **88309777.6**

(22) Date of filing : **19.10.88**

(54) Method and apparatus using variable ranges to support symbolic debugging of optimized code.

(30) Priority : **16.11.87 US 121311**

(43) Date of publication of application : **24.05.89 Bulletin 89/21**

(45) Publication of the grant of the patent : **14.06.95 Bulletin 95/24**

(84) Designated Contracting States : **DE FR GB IT**

(56) References cited :
EP-A- 0 229 245
PROCEEDINGS OF THE SIGPLAN '82 SYMPOSIUM ON COMPILER CONSTRUCTION, Boston,Massachusetts, 23rd - 25th June 1982, pages 98-105; G.J. CHAITIN: "Register allocation & spilling via graph coloring"
W. WAITE et al.: "Compiler construction", chapter 13: "Optimization", 1984,pages 326-357, Springer-Verlag, New York, Inc., US

(73) Proprietor : **Hewlett-Packard Company
Mail Stop 20 B-O,
3000 Hanover Street
Palo Alto, California 94304 (US)**

(72) Inventor : **Meloy, Sue Ann
3256 Todd Way
San Jose California 95124 (US)**
Inventor : **Coutant, Deborah S.
6933 Dartmorr Way
San Jose California 95129 (US)**

(74) Representative : **Colgan, Stephen James et al
CARPMAELS & RANSFORD
43 Bloomsbury Square
London WC1A 2RA (GB)**

Description

Background

The present invention is concerned with debugging code which has been compiled using an optimizing compiler.

Code is generally written in a high level programming language. This high level language, often referred to as source code, is translated by a compiler program into an assembly language. The binary form of the assembly language, called object code is the form of the code actually executed by a computer.

Code debuggers are programs which aid a programmer in finding errors in code. They are extremely useful tools for improving the efficiency of the code debugging process. Many code debuggers supply information pertaining to operation of the code on the assembly code level. If the original code is written in a higher level language, however, this makes program debugging a difficult operation. When a programmer writes his program in a high level language, he does not want to search for the appearance of these errors in the assembly code.

To avoid this problem, it is desirable to develop debugger programs which allow a programmer to debug his program with reference to the level of code in which he originally wrote the program. Such a debugger program is often called a source-level debugger.

One of the important features of a code debugger is to allow a programmer to stop the execution of code and to check the values in each user resource the code is operating upon. A user resource is typically a variable defined in the source code. The values in the user resources give clues to indicate the source of trouble when a program is not operating correctly.

Since the computer operates on object code, a source level debugger needs to know where user resources named in the source code are actually stored by the computer during operation, so that when a user requests the current value for a user resource, the debugging program knows where to find the user resource. Typically, a compiler will allocate a storage location for the user resource where the user resource is always stored. In this case, the debugger need simply go to the location and access the value of user resource.

Difficulties arise when compilers are used which generate optimized code. Usually the design goal of an optimizer within a compiler, is to generate code that executes as fast as possible. In optimized code it may be desirable not to store a user resource in the same place all the time. For instance, if a user resource is accessed often and/or modified often in a particular section of code, during execution of that particular section of code the current value of a user resource may be stored in a register which is accessed and updated, without concurrent update of any other storage location. A standard debugging program, therefore, would have no way of accessing this user resource during execution of the particular section. Thus a programmer is forced to return to attempting to debug code based on the object code, or to use some other method.

EP-A-0229245 describes a technique for allocating and optimizing register assignments during compiling of source code into assembly language. The allocating and optimizing is local to regions of code without branches termed "basic blocks". Each basic block has statements defining computations and each processor involved has memory for storing sequences of executable code and data with means for accessing the memory and executing accessed code. The memory is mapped as a two-level model with a finite number p of registers and a comparably infinite internal memory. The technique comprises ascertaining the data dependency graph attributes of a basic block and generating an allocation and assignment for q of the p registers with reference to all computations within the basic block, by performing a "two-colour pebbling game" heuristic over the ascertained data dependency graph utilizing the two-level memory.

Summary of the Invention

In accordance with the preferred embodiment of the present invention a method and apparatus are presented, as specified in claims 4 and 1, respectively, hereinafter, which uses variable ranges to support symbolic debugging of optimized code. A debug symbol table is constructed which includes descriptions of each user resource in source code.
Additionally, a range table is constructed. The range table contains, for each user resource which is stored in numerous locations during execution of the code, a list of ranges and a description of where the user resource may be found during each range. If the user resource is stored as a constant during a particular range, the value of the constant may be stored in the range table. The description of each user resource in the debug symbol table includes a flag which indicates whether there is a list of ranges in the range table for a particular user resource. If there is, the description of the particular user resource will include a pointer to the list of ranges for that user resource. The pointer may , for example, be an index into the range table for the appropriate list.

Brief Description of the Drawings

Figure 1A shows a computing system in which source code is compiled into object code before being run on a computer.

Figure 1B shows the computing system of Figure 1A with the addition of a debugger.

Figure 2A shows a prior art structure which would be found within a data debugger table such as that shown in Figure 1B.

Figure 2B shows the structure disclosed in Figure 2A modified in conformance with the preferred embodiment of the present invention.

Figure 3 shows a range table in accordance with the preferred embodiment of the present invention.

Figure 4 shows a section of non-optimized code generated by a compiler.

Figure 5 shows the section of code shown in Figure 4 after a code optimizer has performed register promotion.

Figure 5A shows a resource table in accordance with the preferred embodiment of the present invention.

Figure 5B shows the resource table of Figure B after certain user resources have been promoted to registers.

Figure 6 shows the section of code shown in Figure 5, being used to build an interference graph.

Figure 7 shows the section of code shown in Figure 6 after calculation of ranges in accordance with the preferred embodiment of the present invention.

Figure 7A shows a range table which could be developed as a result of the ranges determined in Figure 7 in accordance with the preferred embodiment of the present invention.

Figure 8 shows the section of code shown in Figure 7 after an optimizer has performed copy eliminations.

Figure 9 shows the section of code shown in Figure 8 after ranges for constants have been spilled.

Figure 9A shows the range table shown in Figure 7A after it has been modified to take into account the changes to the codes as recorded in Figure 9 in accordance with the preferred embodiment of the present invention.

Figure 10 shows the section of code shown in Figure 9 after registers have been assigned.

Figure 10A shows the range table shown in Figure 9A after it has been modified to take into account the changes to the code as recorded in Figure 10 in accordance with the preferred embodiment of the present invention.

Figure 11 shows the section of code shown in Figure 10 after a scheduler has switched the location of execution of two instructions.

Figure 11A shows the range table shown in Figure 10A after it has been modified to take into account the changes to the code as recorded in Figure 11 in accordance with the preferred embodiment of the present invention.

Description of the Preferred Embodiment

In Figure 1A, a compiler 42 receives source code 41 and produces object code 43. The object code is run on a computer 44.

In Figure 1B, in order to discover errors in source code 41, a debugger 45 running on computer 44 executes object code 43. During debugging, debugger 45 accesses a debug symbol table 55. Compiler 42 is shown to include a code generator 51 which generates generated code 52. A code optimizer 53 within compiler 42 optimizes generated code 52 to produce object code 43. Code optimizer 53 is a multi-pass global optimizer. Code optimizer 53 keeps track of memory locations and registers through a resource table 54. Each memory location and register has a unique entry in resource table 54. Herein, memory locations are sometimes referred to as memory resources and registers are sometimes referred to a register resources, and memory locations and registers are sometimes collectively referred to as resources.

For each user resource within source code 41, debugger 45 keeps in debug symbol table 55 a data structure. These data structures are entries within debug symbol table 55. A prior art format of the data structure is shown in Figure 2A. In Figure 2A a data structure 106 is comprised of a thirty-two bit word 100, a thirty-two bit word 101, a thirty-two bit word 102, a thirty-two bit word 103 and a thirty-two bit word 104.

Word 100 includes various fields which indicate information about the user resource. A one-bit field 107 stores information which informs the debugger as to size parameters of structure 106. A ten-bit field 108 indicates to debugger 45 what type of structure structure 106 is. For instance, field 108 may indicate to debugger 45 that structure 106 contains a dynamic variable. A one-bit field 109 indicates whether the resource defined by structure 106 is globally recognized in source code 41. A one-bit field 110 indicates whether certain addresses are indirect or direct. A one-bit field 111 indicates whether the resource is a register type, or a dynamic variable. Eighteen bit field 112 is unused.

Word 101 contains a pointer, for instance an actual address or an index, which points to a location in memory in which is kept a name for the resource. The name is how the resource is addressed in source code 41. Word 102 contains a pointer indicating where in memory the resource is stored. Word 103 points to a location in which is stored information about the type of resource. Word 104 contains a post indirection byte offset for use in COBOL structures.

In Figure 2A, debugger 45 always expects to find the resource defined by structure 106 in the same place. As discussed before, this is sufficient only if compiler 42 is not an optimizing compiler.

Figure 2B shows prior art structure 106 modified in accordance with the preferred embodiment of the present invention. Word 101, word 102 word 103 and word 104 remain unchanged. In word 100, a one-bit field 213, and a one-bit field 214 have been added. Field 213 is a pad so that field 214 is in a position that is more convenient for access by debugger 45. Field 214 is reset when there is no range table associated with the resource defined by structure 106. Field 214 is set when there is a range table associated with the resource defined by structure 106.

Additionally, in Figure 2B, a word 205 is added. When field 214 is set, word 205 contains an index into a range table. The index points to the first range for the resource defined by structure 106.

Figure 3 shows a range table in accordance with a preferred embodiment of the present invention. Pointers 311 from debug symbol table, for user resources i and j, point to ranges within the range table. A column 312 contains low bounds for ranges for user resources i and j. A column 313 contains high bounds for ranges for user resources i and j. A column 314 locations for user resources i and j. A column 315 contains flags indicating information about each range.

For instance, a range 301, a range 302 and a range 303 give information about a user resource "i". From program location 4 to program location 100, user resource i is stored as a register resource in register 5. From program location 120 to program location 136, user resource i is stored as a constant with a value of 500. From program location 200 to program location 204 user resource i is is stored as a register resource in register 31. This is the last range for user resource i. For the program locations not given in the range table, the value for user resource i is unavailable.

Similarly, a range 304 and a range 305 given information about a user resource "j". From program location 8 to program location 2C, user resource j is stored as a register resource in register 6. From program location 100 to program location 200 user resource j is stored in a memory stack at location -80 on the stack. This is the last range for user resources j. For the program locations not given in the range table, the value for resource is unavailable.

If therefore data structure 106 were describing user resource i, then field 214 would be set and word 205 would contain an index pointing to range 301. If the data structure 106 were describing user resource j, then field 214 would be set and word 205 would contain an index pointing to range 304. Field 112 has been reduced to sixteen unused bits.

The following is an explanation of how ranges are calculated when code optimizer 53 optimizes generated code 52. In Figure 4 is shown an example of generated code produced by code generator 51. The code in Figure 4 contains three user resources: user resource x, user resource y, and user resource z. It is a goal of this discussion to show how ranges for these user resources may be calculated when code optimizer 53 optimizes generated code 52.

Figure 5 shows generated code 52 of Figure 4 after code optimizer makes a register promotion pass. The code is broken up into four basic blocks: a block 501, a block 502, a block 503 and a block 504. A basic block is a straight-line code sequence with a single entry point and a single exit point.

During the register promotion pass, code optimizer calculates webs for each register or memory location used by generated code 52. What is meant by a web is a set of instructions which access a particular resource. The web begins when the resource is defined to have a particular value. Every successive instruction that uses the resource is included in the web, until the resource is defined to have a new value. More than one web may be built upon a single resource. This occurs when instructions which define and use the resource are completely disjoint from other instructions which define and use the resource.

Since code runs faster when resources are stored in registers, rather than memory locations, on this pass code optimizer looks for memory resources which can be promoted to register resources. As is apparent from close comparison of Figure 5 with Figure 4, user resources x, y and z are promoted from memory resources to registers r9, r10 and r11 respectively. As can be seen in Figure 5 each store to user resource x is changed into a copy to register r9, each store to user resource y is changed into a copy to register r10 and each store to resource z is changed into a copy to register r11.

The webs generated with regard to the code in Figure 5 are shown in the following Table 1:

Table 1

Web Number	Instructions	Resource	User Resource Pointer
web1	1,2	r1	Null
web2	2,6,15	r9	x
web3	3,4	r2	Null
web4	4,7	r10	z
web5	6,8	r3	Null
web6	7,8	r4	Null
web7	8,12,16,17	r5	Null
web8	10,12	r11	y
web9	15,16	r3	Null
web10	16,19	r6	Null
web11	17,19	r9	x

It is an object of the present invention to keep track of which webs are concerned with user resources. This information is stored in the form of a field in each resource table entry within resource table 54. For every resource in resource table 54, if there is a user resource associated therewith, the pointer will point to the entry within debug symbol table 55 for the user resource. If there is not a user resource associated with a resource table entry, the pointer will be null. In Table 1 this pointer for each web is represented by the Column labelled "User Resource Pointer".

In Figure 5A, a portion of resource table 54 is shown. An entry 510 for user resource x includes a pointer 510a which points to an entry for user resource x in debug symbol table 55. An entry 511 for user resource y includes a pointer 511a which points to an entry for user resource y in debug symbol table 55. An entry 512 for user resource z includes a pointer 512a which points to an entry for user resource z in debug symbol table 55.

Figure 5B shows resource table 54 after the register promotion pass. Entries 510, 511 and 512 remain in resource table 54 unchanged. A new entry 520, a new entry 521 and a new entry 522 have been added to resource table 54. Entry 520 for register r9 includes pointer 510a which points to the entry for user resource x in debug symbol table 55. Entry 521 for register r11 includes pointer 511a which points to the entry for user resource y in debug symbol table 55. Entry 522 for register r10 includes pointer 512a which points to the entry for user resource z in debug symbol table 55.

In addition, during the register promotion pass, code optimizer 53 calculates the set of webs which are alive at the end of each of the basic blocks. What is meant by a web being alive at the end of a particular basic block, is that the current value of the resource associated with the web is need by other basic blocks which are executed subsequent to the particular basic block. For the code in Figure 5, the following Table 2 shows which webs are alive at the end of blocks 501, 502, 503 and 504.

Table 2

Basic Block	Live Webs
Block 501	web2, web7
Block 502	web7
Block 503	web7, web10
Block 504	web10

Code optimizer 53 makes another pass to determine interferences between webs. The results of the pass are shown in Figure 6. In Figure 6, each resource is replaced by the web associated with it in Table 1. Additionally, for each instruction the local live web set (llws) is calculated for each instruction. The llws for each instruction is that set of webs which is currently active during the execution of that instruction. The llws for each instruction is calculated as follows. Code optimizer 53 traverses instructions in each of blocks 501-504 in reverse order. The llws for the last instruction in each block is the set of live webs given in Table 2. While traversing the instructions in each block in reverse order, whenever code optimizer 53 encounters a web not in the current llws, code optimizer 53 adds the web to the 11ws. When code optimizer 53 comes across a definition for a web, the web is removed from the llws. The llws which exists after each instruction is processed is shown in Figure 6.

From the information derived during this pass it is determined interferences between webs, that is, which webs are alive during the time other webs are alive. The following Table 3 shows web interferences for the code in Figure 6.

Table 3

Web	Interfering Webs
web1	
web2	web3, web4, web5, web6, web7
web3	web2
web4	web2, web5
web5	web2, web4
web6	web2,web5
web7	web2, web9, web10
web8	
web9	web7
web10	web7
web11	

The same information which code optimizer 53 uses to build the interference graph, may also be used to calculate information on ranges. Ranges may be calculated as follows.

In Table 1 certain webs have pointers to a user resource entry in debug table 55. Information about these webs, are used to calculate ranges for their associated user resources. In calculating these ranges high-water marks and low water marks are used. A high water mark is the instruction before a high bound of a range. A low water mark is the instruction before a low bound of a range.

If at the end of a block a web is still in the llws, the last instruction is a high water mark for the user resource associated with that web. Also, at the point where, code optimizer 53 adds a web to the llws, while traversing the instructions in each block in reverse order, this instruction also is considered a high water mark for the user resource associated with that web. Similarly, when a web is removed from the llws, that instruction is considered a low water mark for the user resource associated with that web. Also the top of a block may be considered a low water mark for any user resource. The above-given system can be improved when the blocks themselves (in addition to the code within each block) are traversed in reverse order. In this case, if a web is in the llws at the end of the next-to-be traversed block, it is not necessary to place a low water mark at the top of the most recently traversed block.

Using Figure 6 and Figure 7 it is possible to see how the method of determining ranges described above, may be performed on the code shown in Figure 6. Starting with instruction 19, web11 is in the llws. As can be seen from Table 1, web11 has a user resource pointer to user resource x. Therefore, instruction 19 is a high water mark for user resource x. In instruction 17, web11 is defined, and therefore removed from the 11ws. Therefore, instruction 17 is a low water mark for user resource x. A high bound and a low bound for Range 701 for user resource x may be determined from the high water mark and the low water mark given.

In instruction 15, web2 is used and it is not currently in the llws, therefore it is the last use of web2. From Table 1, web2 has a user resource pointer to user resource x. Therefore, instruction 15 is a high water mark for user resource x. Instruction 14 is the top of block 503. Web2 is not in the 11ws at the end of block 502. Therefore, instruction 14 is a low water mark for user resource x. Range 702 for user resource x may be determined from the high water mark and the low water mark.

In instruction 12, web8 is used and it is not currently in the llws, therefore it is the last use of web8. From Table 1, web8 has a user resource pointer to user resource y. Therefore, instruction 12 is a high water mark for user resource y. In instruction 10, web8 is defined. Therefore instruction 10 is a low water mark for user resource y. Range 703 for user resource x may be determined from the high water mark and the low water mark.

Instruction 9 is the end of block 501. Web2 is in the llws in instruction 9 at the end of block 501. From Table 1, web2 has a user resource pointer to user resource x. Therefore, instruction 12 is a high water mark for user resource x.

In instruction 7, web4 is used and it is not in the llws, therefore it is the last use of web4. From Table 1, web4 has a resource pointer to user resource z. Therefore, instruction 7 is a high water mark for user resource z. In instruction 4, web4 is defined. Therefore, instruction 4 is a low water mark for user resource z. Range 704 for user resource z may be determined from the high water mark and the low water mark.

In instruction 2, web2 is defined. Therefore, instruction 2 is a low water mark for user resource x. Range 705 for user resource x may be determined from the high water mark and the low water mark. At this point a range table could be constructed, as is shown in Figure 7A.

In the range table, in Figure 7A pointers 131 from debug symbol table, for user resources x, y and z, point to ranges within the range table. A column 132 contains low bounds for ranges for user resources x, y and z. A column 133 contains high bounds for ranges for user resources x, y and z. A column 134 contains a location for the user resource for the respective ranges. A column 135 indicates the type of resource in which user resource resides during the given range. Information about the ranges are associated with the respective webs. For instance, this may be done by adding a list of ranges to a data structure built and used by a portion (called a register allocator) of code optimizer 53 which allocates registers. The data structure built by the allocator is called an interference graph. Each web has a separate entry. The list of ranges for each web is added to the web's entry in the interference graph.

When code optimizer 53 makes additional passes in the code, the ranges may need to be adjusted accordingly. For instance, code optimizer 53 may perform a copy elimination. A copy elimination may be performed when code optimizer 53 determines that two webs do not interfere, but share a copy instruction in common. The copy is removed and the two webs become a single composite web.

If a web is removed which is associated with a user resource, the list of ranges for the removed web must be added to the composite web. Figure 8 shows the code of Figure 7 after code optimizer 53 performs copy elimination. The following Table 4 shows the necessary modifications to Table 1, resulting from the copy elimination:

Table 4

Web Number	Instructions	Resource	User Resource Pointer
web1:web2			
web2:web9			
web3:web4			
web4:web6			
web5	6,8	r3	Null
web6	7,8,3,4,4,7	r4, r2, r10	z
web7:web11			
web8:web7			
web9	15,16,2,6, 15,1,2	r3, r9, r1	x
web10	16,19	r6	Null
web11	17,19,8,12, 16,17,10,12	r9, r5, r11	x,y

Code optimizer 53 may also perform a constant spill. For instance, if a web contains only a constant variable, code optimizer 53 may determine that it is advantageous to free up a register by reloading the constant into a register before each use, rather than leaving the constant continually in the register. In this case the ranges, which previously were of type register range, are changed to type constant range. The location of the user resource is changed to equal the constant. Figure 9 shows how the code of Figure 8 is modified after code optimizer 53 performs a constant spill. Figure 9A shows how the range table showed in Figure 7A would be modified as a result of the constant spill.

At this point code optimizer 53 allocates registers to webs. Figure 10 shows the code of Figure 9 after registers have been allocated. Once registers are allocated to webs, the information about those registers may be filled into the range table. Figure 10A shows the range table of Figure 9A after register allocation.

If, during the register allocation, there are not sufficient registers, code optimizer 53 spills some of the webs to memory. A spilled web is stored to memory after being defined, and reloaded before it is used. When this occurs entries in the range table are changed to spill ranges. The low water mark and high water mark remain the same, but the location is changed to the spill memory location and the type of range is changed to spill.

The ranges can often be expanded by making another pass. The high water mark may be moved to the end of a basic block or to an instruction that might wipe out the contents of the register in which the user resource resides, whichever first occurs.

In order to do this, the code is again traversed in reverse order, keeping track of the last instruction of the basic block as an extend instruction for each web. A set of flags keeps track of the registers defined between the current instruction and the last instruction of the basic block. When a definition of a register is seen, the flag for the appropriate register is set and a pointer to the defining instruction is attached to the web. The defining instruction becomes a new extend instruction for that web.

When an instruction performing a call is seen, the call instruction becomes the extend instruction in all the webs. This is because a call could wipe out the values in any register.

When the use of a web is seen, if the register assigned to that web had been defined before the last instruction, that instruction becomes the extend instruction for that web.

When the traverse is finished, for those webs which contain a range whose high water mark is lower than the extend instruction, the extend instruction becomes the new high water mark, and the range is adjusted accordingly.

Finally, additional functional units within code optimizer may operate on the code to produce ranges which overlap existing ranges. In this case the range with the lowest priority must be shortened. For example, in Figure 11, a scheduler within code optimizer 53 has reversed the operations of instructions 11 and instruction 10. This has created a scheduler range 119 which overlapped range 703. Range 703 was created first, so has the lowest priority. Therefore, range 703 was shortened. Figure 11A shows how the range table shown in Figure 10A is modified to reflect the shortening of range 703.

Claims

1. A computing system which includes a debugger (45) which is used to debug compiled code (43) generated by a compiler (42) from source code (41), wherein the debugger (45) accesses a debug symbol table (55) containing a plurality of entries, each entry (2B) in the plurality of entries describing user resources used by the source code, said computing system additionally comprising:
   a range table (312-313) accessible by the debugger (45) including
   specification of locations (314) where at least some user resources reside in the computing system when the computing system executes the compiled code (43), and
   specification of ranges (312,313) within the compiled code when the user resources reside in each of the specified locations,
   wherein when a list of ranges (312,313) in the range table specify locations of a first user resource, a first entry from the plurality of entries in, said symbol debug table which describes the first user resource includes a pointer (311) which identifies the list of ranges.

2. A computing system according to claim 1
   wherein the first plurality of entries includes a first subset of entries (705,702,704 (Figures 9A)) which describe user resources for which the range table does specify locations and includes a second subset of entries (701,703 (Figure 9A)) which describe user resources for which the range table does not specify locations; and
   wherein, each entry within the first plurality of entries contains a field (134) which indicates whether the entry is in the first subset of entries or the second subset of entries.

3. A computing system according to claim 2 wherein each entry in the second set of entries contains a pointer (135) which indicates a location in the computing system where a user resource described by the entry resides when the computing system executes the compiled code.

4. A method useful for calculating, during optimization of code, ranges for which user resources reside in certain locations within a computing system, the method comprising:
   (a) dividing the code up into basic blocks (501,502,503,504) of straight line code sequences with a single entry point and a single exit point;
   (b) calculating webs for each user resource, a web being a set of instructions which access a particular resource;
   (c) determining the range of instructions (132,133) within each basic block when webs for each user resource is alive;
   (d) allocating, for each determined range of instructions for each web, locations in which user resources are stored; and,
   (e) storing in a range table a description of allocated locations (135) for each determined range.

5. A method according to claim 4, wherein after step (c) the determined range of instructions for some webs is extended by the following steps:
(c1) combining a first range of instructions for a first web with a second range of instructions for the first web when the first range of instructions is in a first basic block of code and the second range of instructions is in a second basic block of code which immediately follows the first block of code, and when the first web is alive at the last instruction of the first block and is alive at the first instruction of the second block, and wherein the first range of instructions and the second range of instructions are combined into a third range of instructions which includes all the instructions in the first range of instructions and the second range of instructions.

6. A method according to claim 4 or 5, additionally comprising the step of:
(f) storing in the table a constant value (704) when the user resource is stored as a constant.

7. A method according to claim 4, additionally comprising the step of:
(f) changing the description of the allocated location (134) for a determined range (133,134) of instructions when the location in which the user resources are stored are changed for that determined range of instructions.

8. A method according to claim 4 additionally comprising the step of:
(f) redetermining a first range of instructions which use a first user resource when a second range of instructions uses the first resource and overlaps the first range of instructions, and when the second range of instructions has a higher priority than the first range of instructions.

9. A method according to any of claims 4 to 8 wherein, after step (b), is added the following steps:
(b1) combining a first web for a first user resource with a second web for a second resource into a composite web when the first web and the second web share a copy instruction; but do not interfere with each other;
(b2) substituting the composite web for the first web and for the second web as the calculated web for the first user resource and for the second user resource.

Patentansprüche

1. Ein Berechnungssystem, das ein Fehlersuchprogramm (45) einschließt, das verwendet wird, um einen übersetzten Code (43) von Fehlern zu befreien, der durch einen Compiler (42) aus einem Qellcode (41) erzeugt wurde, wobei das Fehlersuchprogramm (45) auf eine Fehlersuchsymboltabelle (55) zugreift, die eine Mehrzahl von Einträgen enthält, wobei jeder Eintrag (2B) der Mehrzahl von Einträgen Anwenderbetriebsmittel beschreibt, die durch den Quellcode verwendet werden, wobei das Berechnungssystem zusätzlich folgende Merkmale umfaßt:
eine Bereichstabelle (312 - 313), auf die durch das Fehlersuchprogramm (45) zugegriffen werden kann, die folgende Merkmale einschließt:
eine Spezifikation von Orten (314), an denen zumindest einige Anwenderbetriebsmittel in dem Berechnungssystem vorhanden sind, wenn das Berechnungssystem den übersetzten Code (43) ausführt, und
eine Spezifikation von Bereichen (312, 313) innerhalb des übersetzten Codes, wenn die Anwenderbetriebsmittel an jedem der bestimmten Orte vorhanden sind,
wobei, wenn eine Liste von Bereichen (312, 313) in der Bereichstabelle Orte eines ersten Anwenderbetriebsmittels bestimmen, ein erster Eintrag aus einer Mehrzahl von Einträgen in der Fehlersuchsymboltabelle, die das erste Anwenderbetriebsmittel beschreiben, einen Zeiger (311) einschließt, der eine Liste von Bereichen identifiziert.

2. Ein Berechnungssystem nach Anspruch 1, bei dem die erste Mehrzahl von Einträgen eine erste Teilmenge von Einträgen (705, 702, 704 (Figur 9A)) einschließt, die Anwenderbetriebsmittel beschreiben, für die die Bereichstabelle Orte bestimmt, und eine zweite Teilmenge von Einträgen (701, 703 (Figur 9A)) einschließt, die Anwenderbetriebsmittel beschreiben, für die die Bereichstabelle keine Orte bestimmt; und bei dem jeder Eintrag innerhalb der ersten Mehrzahl von Einträgen ein Feld (134) enthält, das angibt, ob der Eintrag in der ersten Teilmenge von Einträgen oder in der zweiten Teilmenge von Einträgen ist.

3. Ein Berechnungssystem nach Anspruch 2, bei dem jeder Eintrag in dem zweiten Satz von Einträgen einen

Zeiger (135) enthält, der einen Ort in dem Berechnungssystem anzeigt, an dem ein Anwenderbetriebsmittel, das durch den Eintrag beschrieben ist, vorhanden ist, wenn das Berechnungssystem den übersetzten Code ausführt.

4. Ein Verfahren, das zur Berechnung von Bereichen während der Optimierung eines Codes nützlich ist, bei denen Anwenderbetriebsmittel an bestimmten Orten innerhalb eines Berechnungssystems vorhanden sind, wobei das Verfahren folgende Schritte aufweist:
(a) Teilen des Codes in Basisblöcke (501, 502, 503, 504) aus geradlinigen Codefolgen mit einem einzelnen Eintrittspunkt und einem einzelnen Austrittspunkt;
(b) Berechnen der Netze für jedes Anwenderbetriebsmittel, wobei ein Netz ein Satz von Befehlen ist, die auf ein bestimmtes Betriebsmittel zugreifen;
(c) Bestimmen des Bereichs von Befehlen (132, 133) innerhalb eines jeden Basisblocks, wenn Netze für jedes Anwenderbetriebsmittel aktiv sind;
(d) Zuteilen von Orten für jeden bestimmten Bereich von Befehlen für jedes Netz, an dem die Anwenderbetriebsmittel gespeichert sind; und
(e) Speichern einer Beschreibung der zugeteilten Orte (135) für jeden bestimmten Bereich in einer Tabelle.

5. Ein Verfahren nach Anspruch 4, bei dem nach dem Schritt (c) der bestimmte Bereich der Befehle für einige Netze durch die folgenden Schritte erweitert wird:
(c1) Kombinieren eines ersten Bereichs von Befehlen für ein erstes Netz mit einem zweiten Bereich von Befehlen für das erste Netz, wenn der erste Bereich von Befehlen in einem ersten Basiscodeblock und der zweite Bereich von Befehlen in einem zweiten Basiscodeblock liegt, der dem ersten Codeblock direkt folgt, und wenn das erste Netz an dem letzten Befehl des ersten Blocks aktiv ist, und wenn es am ersten Befehl des zweiten Blocks aktiv ist, und wobei der erste Bereich von Befehlen und der zweite Bereich von Befehlen in einen dritten Bereich von Befehlen kombiniert wird, der alle Befehle in dem ersten Bereich von Befehlen und in dem zweiten Bereich von Befehlen einschließt.

6. Ein Verfahren nach Anspruch 4 oder 5, das zusätzlich folgenden Schritt aufweist:
(f) Speichern eines konstanten Wertes (704) in der Tabelle, wenn das Anwenderbetriebsmittel als eine Konstante gespeichert ist.

7. Ein Verfahren nach Anspruch 4, das zusätzlich folgenden Schritt aufweist:
(f) Ändern der Beschreibung des zugeteilten Ortes (134) für einen bestimmten Bereich (133, 134) von Befehlen, wenn der Ort, an dem die Anwenderbetriebsmittel gespeichert sind, für diesen bestimmten Bereich von Befehlen geändert wird.

8. Ein Verfahren nach Anspruch 4, das zusätzlich folgenden Schritt aufweist:
(f) Erneutes Bestimmen eines ersten Bereichs von Befehlen, die ein erstes Anwenderbetriebsmittel verwenden, wenn ein zweiter Bereich von Befehlen das erste Betriebsmittel verwendet und den ersten Bereich von Befehlen überlappt, und wenn der zweite Bereich von Befehlen eine höhere Priorität aufweist als der erste Bereich von Befehlen.

9. Ein Verfahren gemäß irgendeinem der Ansprüche 4 bis 8, bei dem nach dem Schritt (b) die folgenden Schritte hinzugefügt werden:
(b1) Verbinden eines ersten Netzes für ein erstes Anwenderbetriebsmittel mit einem zweiten Netz für ein zweites Betriebsmittel zu einem zusammengesetzten Netz, wenn das erste Netz und das zweite Netz einen Kopierbefehl miteinander gemeinsam verwenden; aber nicht miteinander wechselwirken;
(b2) Ersetzen des ersten Netzes und des zweiten Netzes durch das zusammengesetzte Netz als das berechnete Netz für das erste Anwenderbetriebsmittel und für das zweite Anwenderbetriebsmittel.

Revendications

1. Système de calcul comportant un débogueur (45), qui est utilisé pour déboguer un code compilé (43) généré par un compilateur (42) à partir d'un code source (41), dans lequel le débogueur (45) accède à une table de symboles de débogage (55) contenant plusieurs entrées, chaque entrée (2B) parmi les plusieurs entrées décrivant les ressources utilisateur utilisées par le code source, ledit système de calcul comprenant de plus;

une table de plages (312-313) accessible par le débogueur (45) comportant

la spécification des emplacements (314) où au moins certaines ressources utilisateur résident dans le système de calcul lorsque le système de calcul exécute le code compilé (43), et

la spécification des plages (312, 313) situées à l'intérieur du code compilé lorsque les ressources utilisateur résident dans chacun des emplacements spécifiés,

dans lequel, lorsqu'une liste de plages (312, 313) de la table de plages spécifie les emplacements d'une première ressource utilisateur, une première entrée parmi les plusieurs entrées de ladite table des symboles de débogage qui décrit la première ressource utilisateur comporte un pointeur (311) qui identifie la liste de plages.

2. Système de calcul selon la revendication 1,

dans lequel, le premier ensemble d'entrées comporte un premier sous-ensemble d'entrées (705, 702, 704 (Figure 9A)), qui décrit les ressources utilisateur pour lesquelles la table de plages spécifie des emplacements, et comporte un second sous-ensemble d'entrées (701, 703 (Figure 9A)) qui décrit les ressources utilisateur pour lesquelles la table de plages ne spécifie pas d'emplacements; et

dans lequel, chaque entrée du premier ensemble d'entrées contient un champ (134) qui indique si l'entrée se trouve dans le premier sous-ensemble d'entrées ou dans le second sous-ensemble d'entrées.

3. Système de calcul selon la revendication 2, dans lequel chaque entrée du second ensemble d'entrées contient un pointeur (135) qui indique un emplacement dans le système de calcul où réside une ressource utilisateur décrite par l'entrée lorsque le système de calcul exécute le code compilé.

4. Procédé utilisé pour calculer, pendant l'optimisation d'un code, des plages pour lesquelles les ressources utilisateur résident à certains emplacements d'un système de calcul, le procédé comprenant:

(a) la division du code en blocs de base (501, 502, 503, 504) de séquences de code en ligne directe avec un point d'entrée unique et un point de sortie unique;

(b) le calcul des tissus pour chaque ressource utilisateur, un tissu étant un ensemble d'instructions qui accèdent à une ressource particulière;

(c) la détermination de la plage d'instructions (132, 133) dans chaque bloc de base lorsque les tissus pour chaque ressource utilisateur sont vivants;

(d) l'allocation, pour chaque plage déterminée des instructions pour chaque tissu, des emplacements dans lesquels sont stockées les ressources utilisateur; et

(e) le stockage dans une table, de la description des emplacements alloués (135) pour chaque plage déterminée.

5. Procédé selon la revendication 4, dans lequel après l'étape (c), la plage déterminée d'instructions pour certains tissus est complétée par les étapes suivantes:

(c1) association d'une première plage d'instructions pour un premier tissu avec une seconde plage d'instructions pour le premier tissu, lorsque la première plage d'instructions se trouve dans un premier bloc de base de code et que la seconde plage d'instructions se trouve dans un second bloc de base de code qui suit immédiatement le premier bloc de code, et lorsque le premier tissu est vivant à la dernière instruction du premier bloc, et qu'il est vivant à la première instruction du second bloc, et dans lequel la première plage d'instructions et la seconde plage d'instructions sont associées en une troisième plage d'instructions qui comporte toutes les instructions de la première plage d'instructions et de la seconde plage d'instructions.

6. Procédé selon la revendication 4 ou 5, comprenant de plus les étapes de:

(f) stockage dans la table d'une valeur constante (704), lorsque la ressource utilisateur est stockée en tant que constante.

7. Procédé selon la revendication 4, comprenant de plus l'étape de:

(f) modification de la description de l'emplacement alloué (134) pour une plage déterminée (133, 134) d'instructions, lorsque l'emplacement dans lequel les ressources utilisateur sont stockées, est modifié pour cette plage déterminée d'instructions.

8. Procédé selon la revendication 4, comprenant de plus l'étape de:

(f) nouvelle détermination d'une première plage d'instructions qui utilise une première ressource utilisateur lorsqu'une seconde plage d'instructions utilise la première ressource, et recouvre la première plage d'instructions, et lorsque la seconde plage d'instructions a une priorité supérieure à celle de la première plage d'instructions.

9. Procédé selon l'une quelconque des revendications 4 à 8, dans lequel, après l'étape (b), sont ajoutées les étapes suivantes:
(b1) association d'un premier tissu pour une première ressource utilisateur avec un second tissu avec une seconde ressource, en un tissu composite, lorsque le premier tissu et le second tissu partagent une instruction de copie; mais n'interfèrent pas entre eux;
(b2) substitution du tissu composite au premier tissu et second tissu, en tant que tissu calculé pour la première ressource utilisateur et pour la seconde ressource utilisateur.

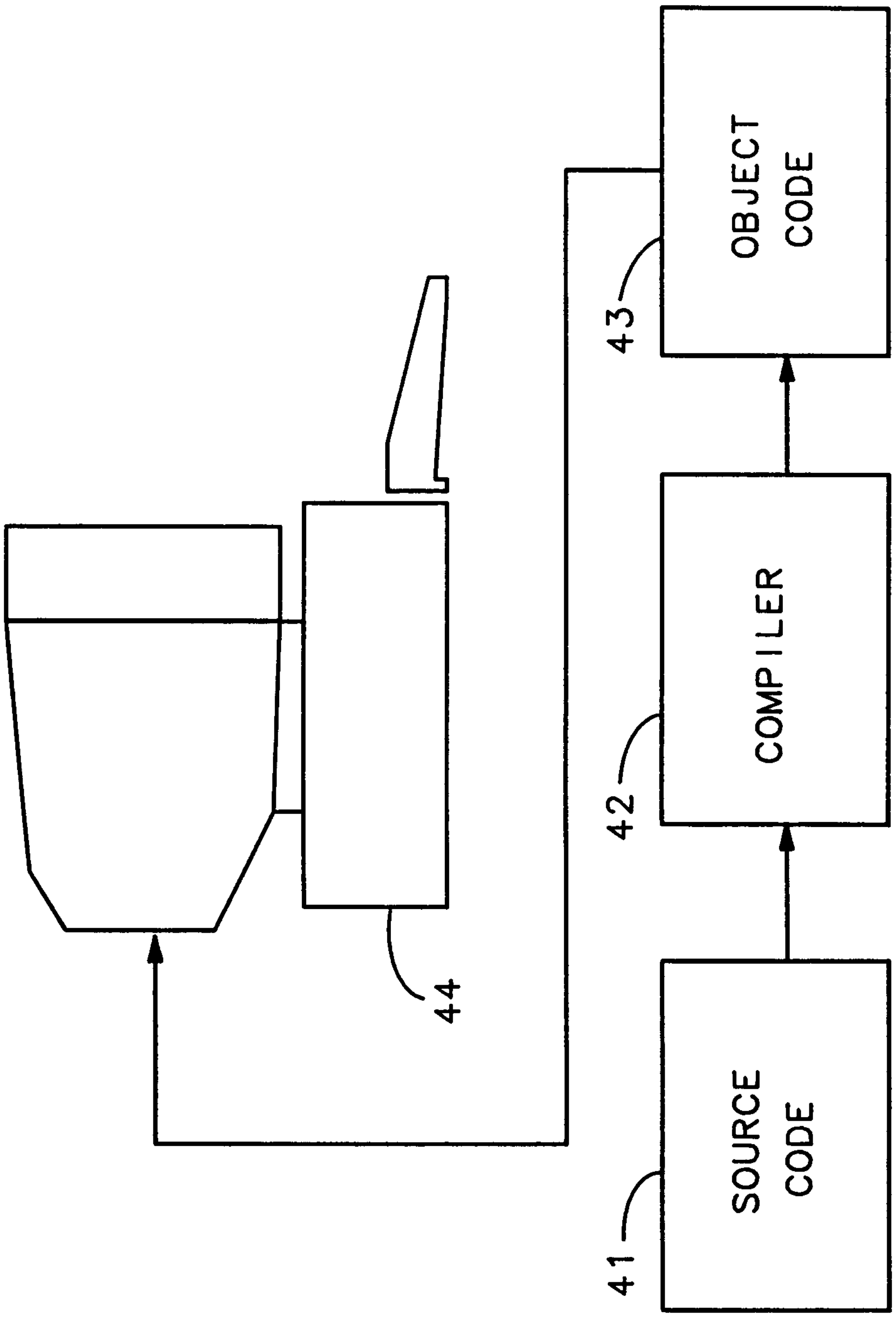


FIG 1A

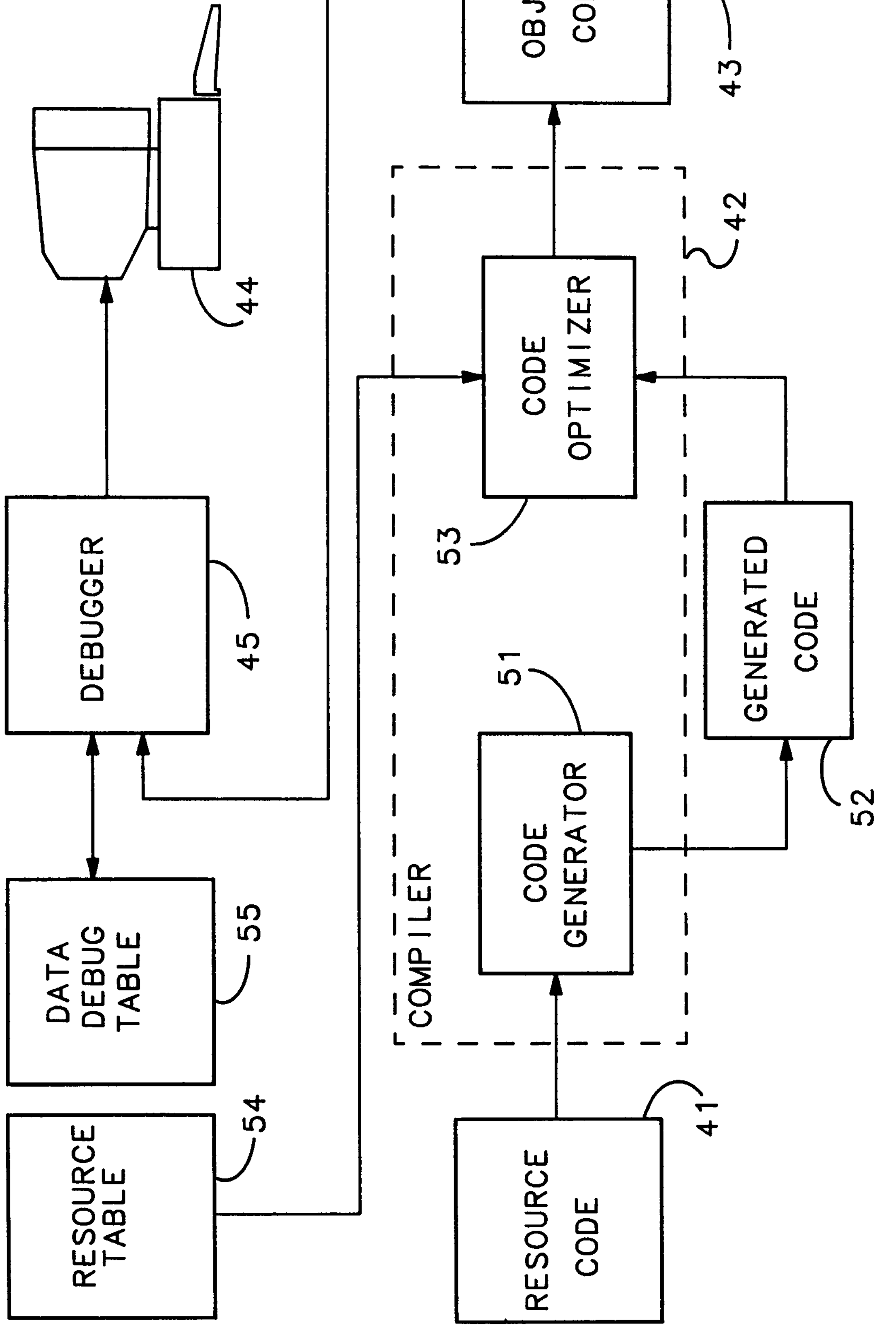
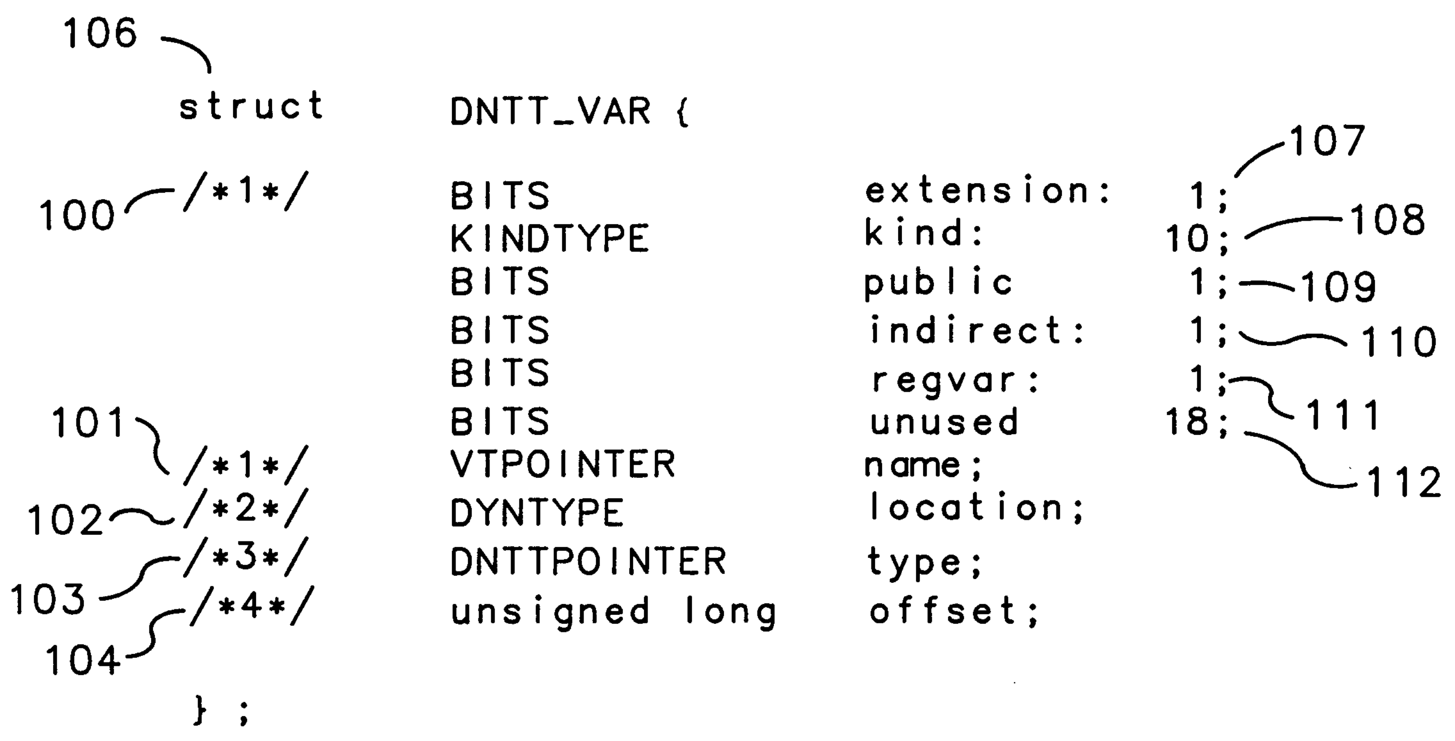

RESOURCE TABLE
54
DATA DEBUG TABLE
55
DEBUGGER
45
44
RESOURCE CODE
41
COMPILER
CODE GENERATOR
51
GENERATED CODE
52
CODE OPTIMIZER
53
42
OBJECT CODE
43

FIG 1B



```
106
    struct      DNTT_VAR {
                                                        107
100 /*1*/       BITS             extension:   1;
                KINDTYPE         kind:       10;        108
                BITS             public       1;        109
                BITS             indirect:    1;        110
                BITS             regvar:      1;        111
101             BITS             unused      18;        112
    /*1*/       VTPOINTER        name;
102 /*2*/       DYNTYPE          location;
103 /*3*/       DNTTPOINTER      type;
104 /*4*/       unsigned long    offset;
    };
```

FIG 2A (PRIOR ART)

```
106
     struct      DNTT_VAR {
                                                  107
100  /*1*/       BITS            extension:   1;
                 KINDTYPE        kind:       10;  108
                 BITS            public       1;  109
                 BITS            indirect:    1;  110
                 BITS            regvar:      1;  111
                 BITS            dummy:       1;  213
                 BITS            range:       1;  214
101              BITS            unused      16;
     /*1*/       VTPOINTER       name;
102  /*2*/       DYNTYPE         location;        112
103  /*3*/       DNTTPOINTER     type;
104  /*4*/       unsigned long   offset;
     /*5*/       unsigned long range_index;
205
     };
```

FIG 2B

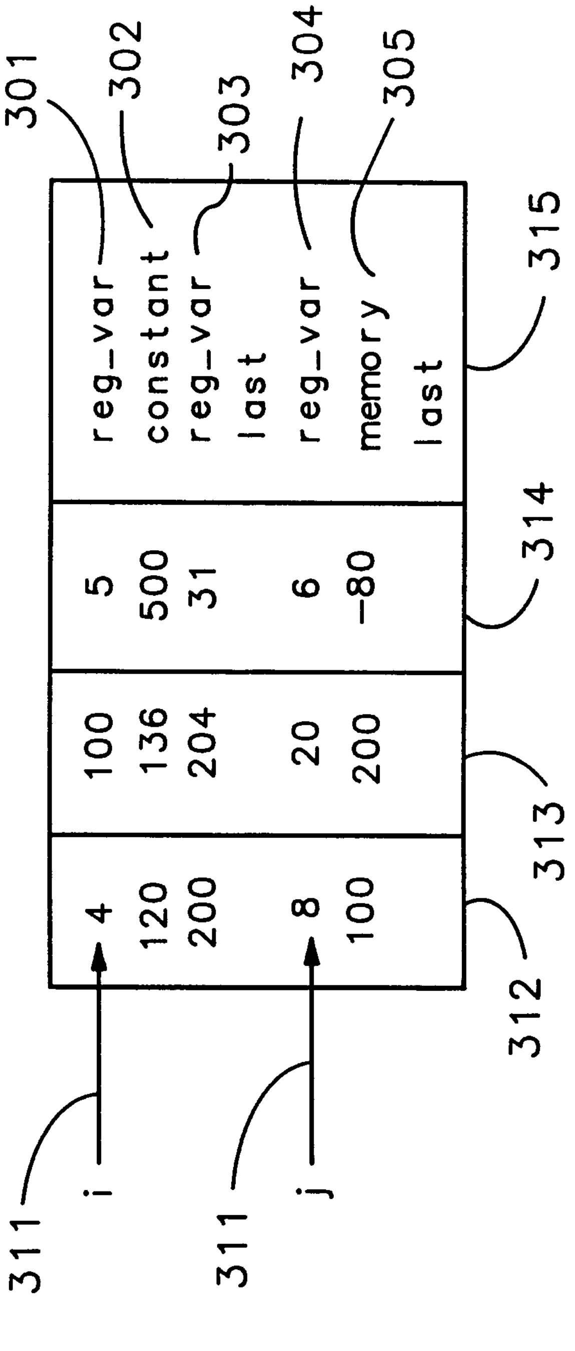
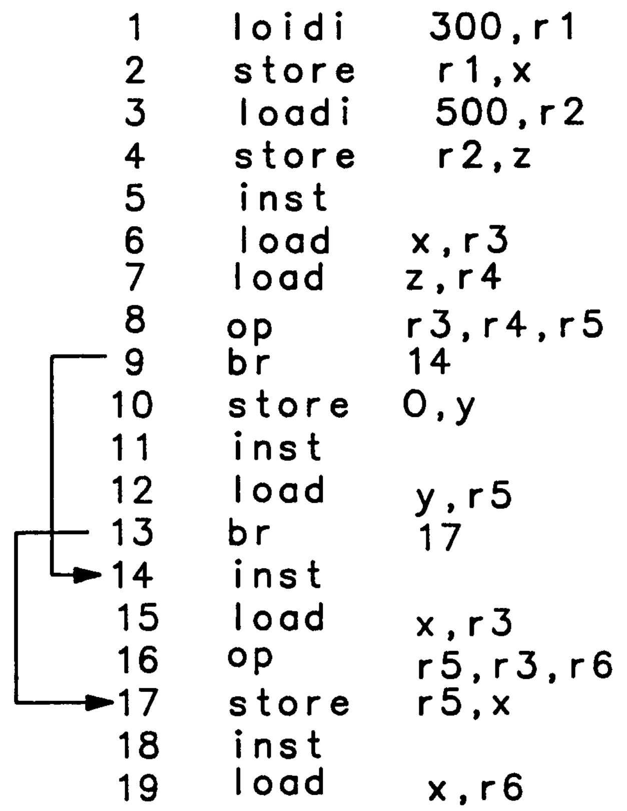
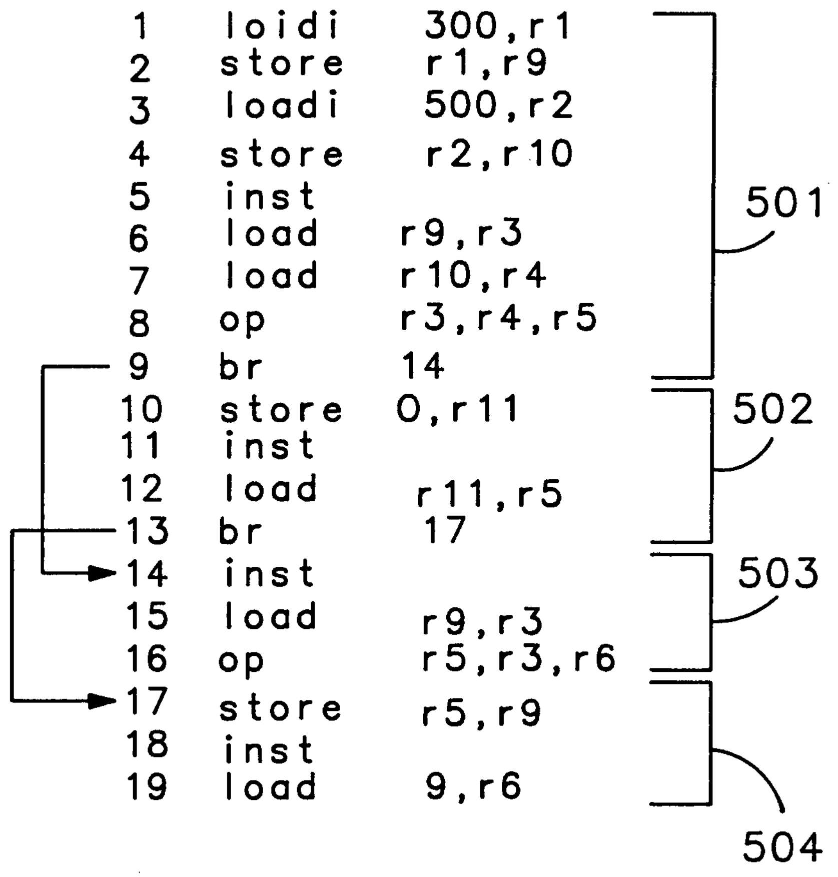

301
302
303
304
305
reg_var
constant
reg_var
last
reg_var
memory
last
5
500
31
6
−80
100
136
204
20
200
4
120
200
8
100
i
j
311
311
312
313
314
315

FIG 3



```
 1   loidi   300,r1
 2   store   r1,x
 3   loadi   500,r2
 4   store   r2,z
 5   inst
 6   load    x,r3
 7   load    z,r4
 8   op      r3,r4,r5
 9   br      14
10   store   0,y
11   inst
12   load    y,r5
13   br      17
14   inst
15   load    x,r3
16   op      r5,r3,r6
17   store   r5,x
18   inst
19   load    x,r6
```

FIG 4



```
 1   loidi   300,r1
 2   store   r1,r9
 3   loadi   500,r2
 4   store   r2,r10
 5   inst
 6   load    r9,r3
 7   load    r10,r4
 8   op      r3,r4,r5
 9   br      14
10   store   0,r11
11   inst
12   load    r11,r5
13   br      17
14   inst
15   load    r9,r3
16   op      r5,r3,r6
17   store   r5,r9
18   inst
19   load    9,r6
```

FIG 5

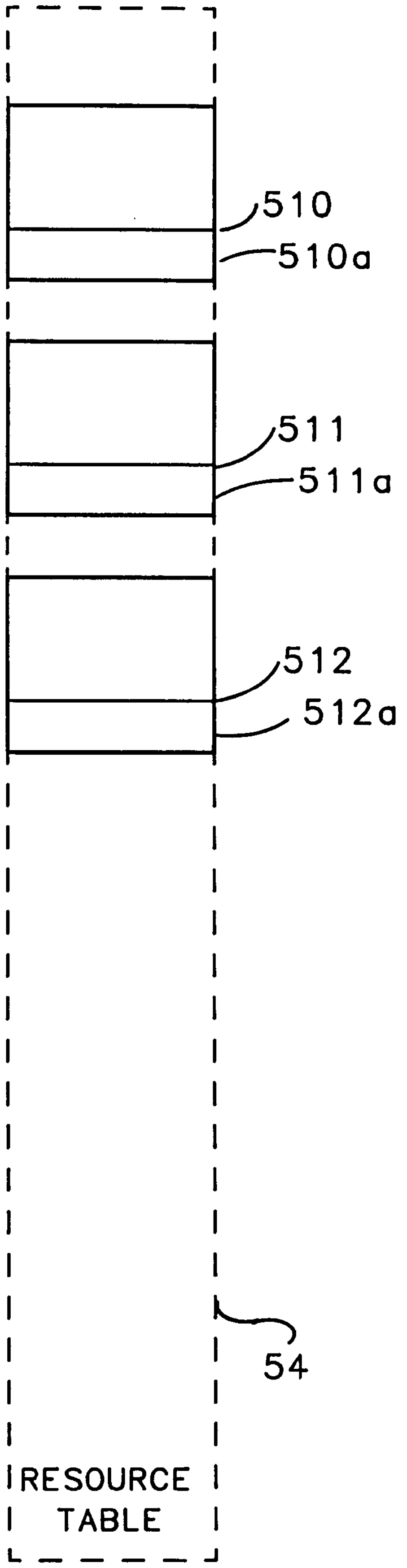

510
510a
511
511a
512
512a
54
RESOURCE TABLE

FIG 5A

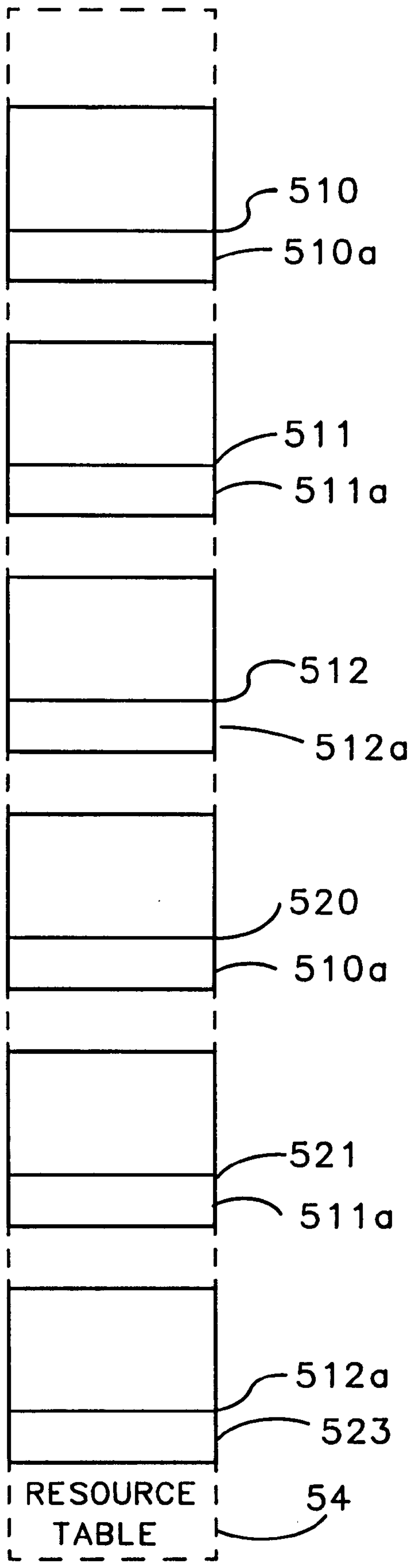
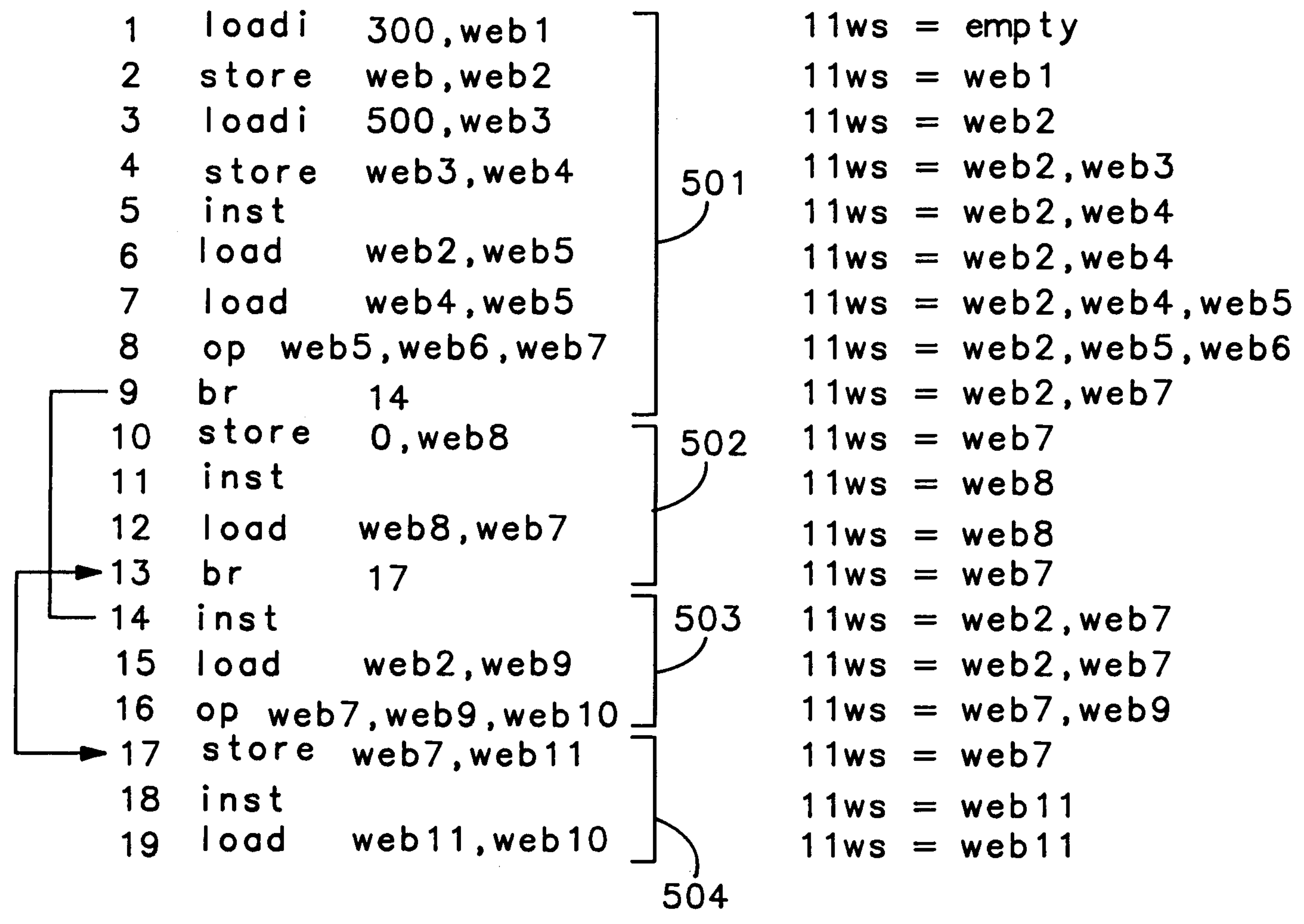
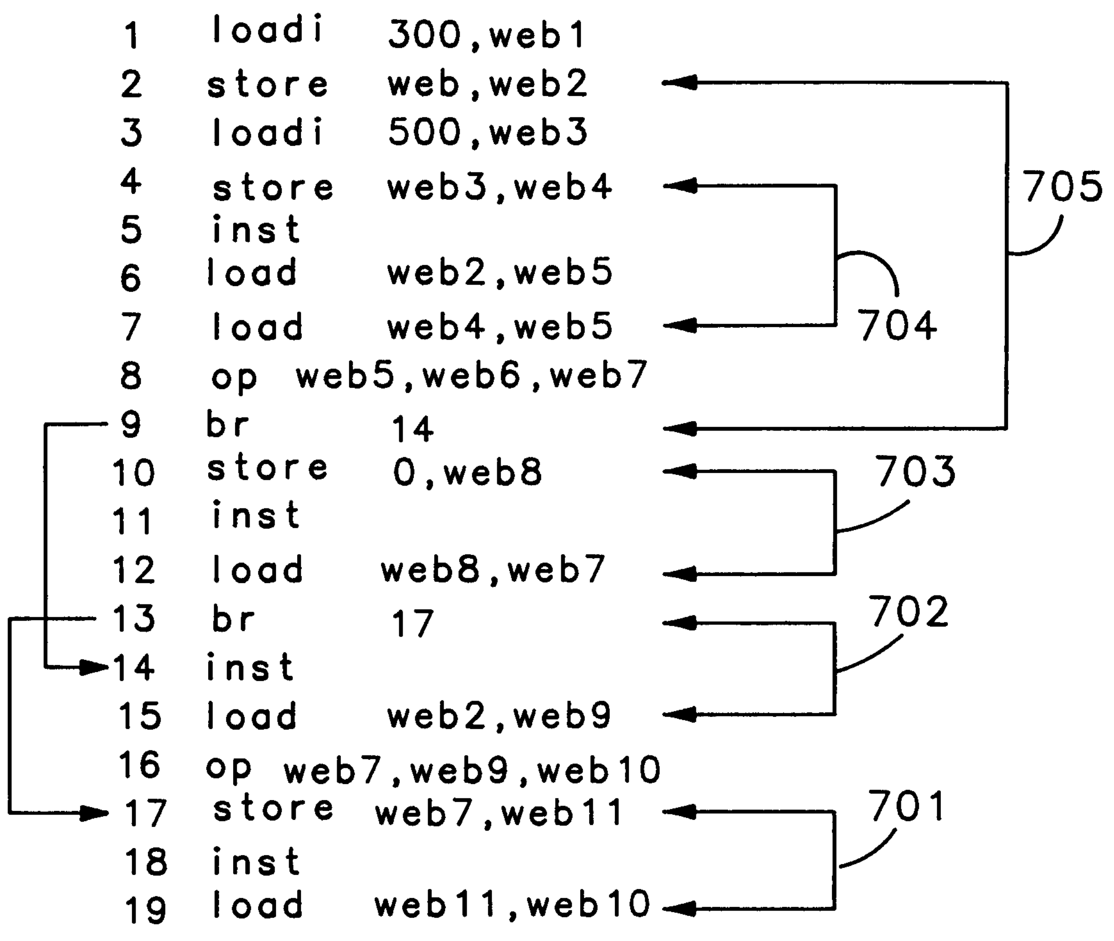

510
510a
511
511a
512
512a
520
510a
521
511a
512a
523
54
RESOURCE TABLE

FIG 5B



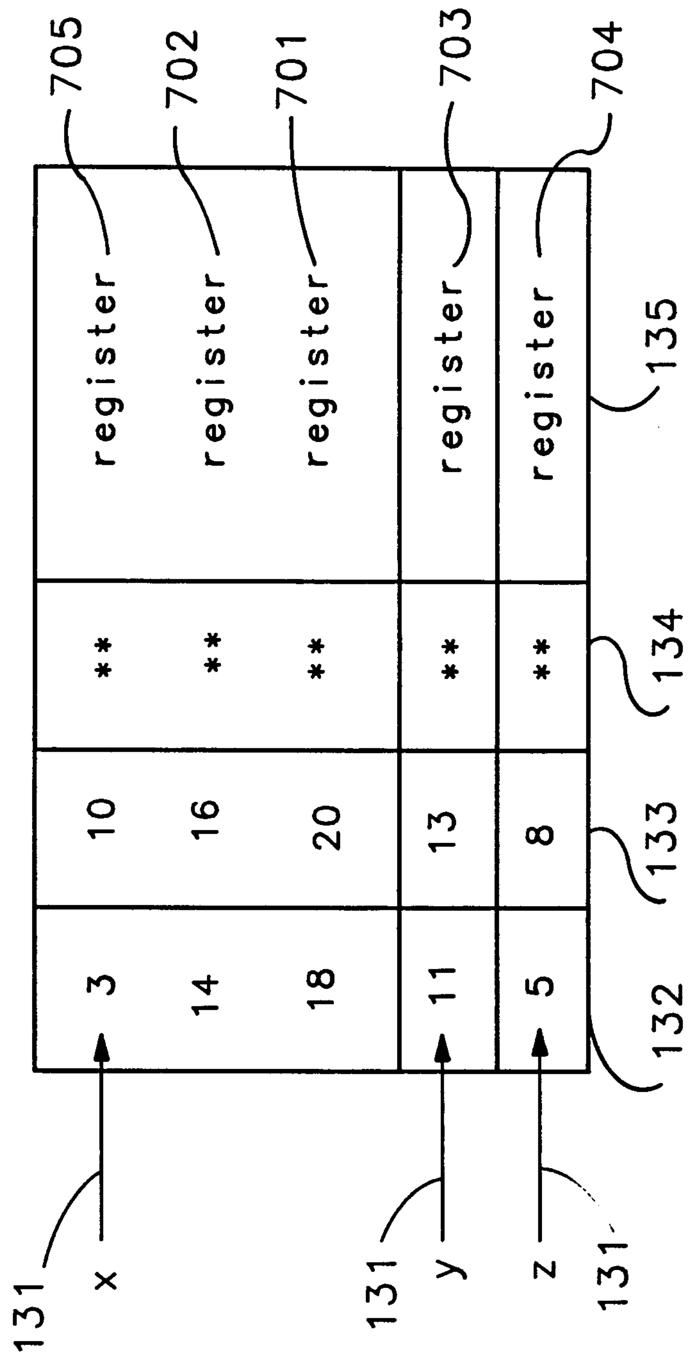
```
1   loadi  300,web1          11ws = empty
2   store  web,web2          11ws = web1
3   loadi  500,web3          11ws = web2
4   store  web3,web4         11ws = web2,web3
5   inst                     11ws = web2,web4
6   load   web2,web5         11ws = web2,web4
7   load   web4,web5         11ws = web2,web4,web5
8   op  web5,web6,web7       11ws = web2,web5,web6
9   br     14                11ws = web2,web7
10  store  0,web8            11ws = web7
11  inst                     11ws = web8
12  load   web8,web7         11ws = web8
13  br     17                11ws = web7
14  inst                     11ws = web2,web7
15  load   web2,web9         11ws = web2,web7
16  op  web7,web9,web10      11ws = web7,web9
17  store  web7,web11        11ws = web7
18  inst                     11ws = web11
19  load   web11,web10       11ws = web11
```

FIG 6



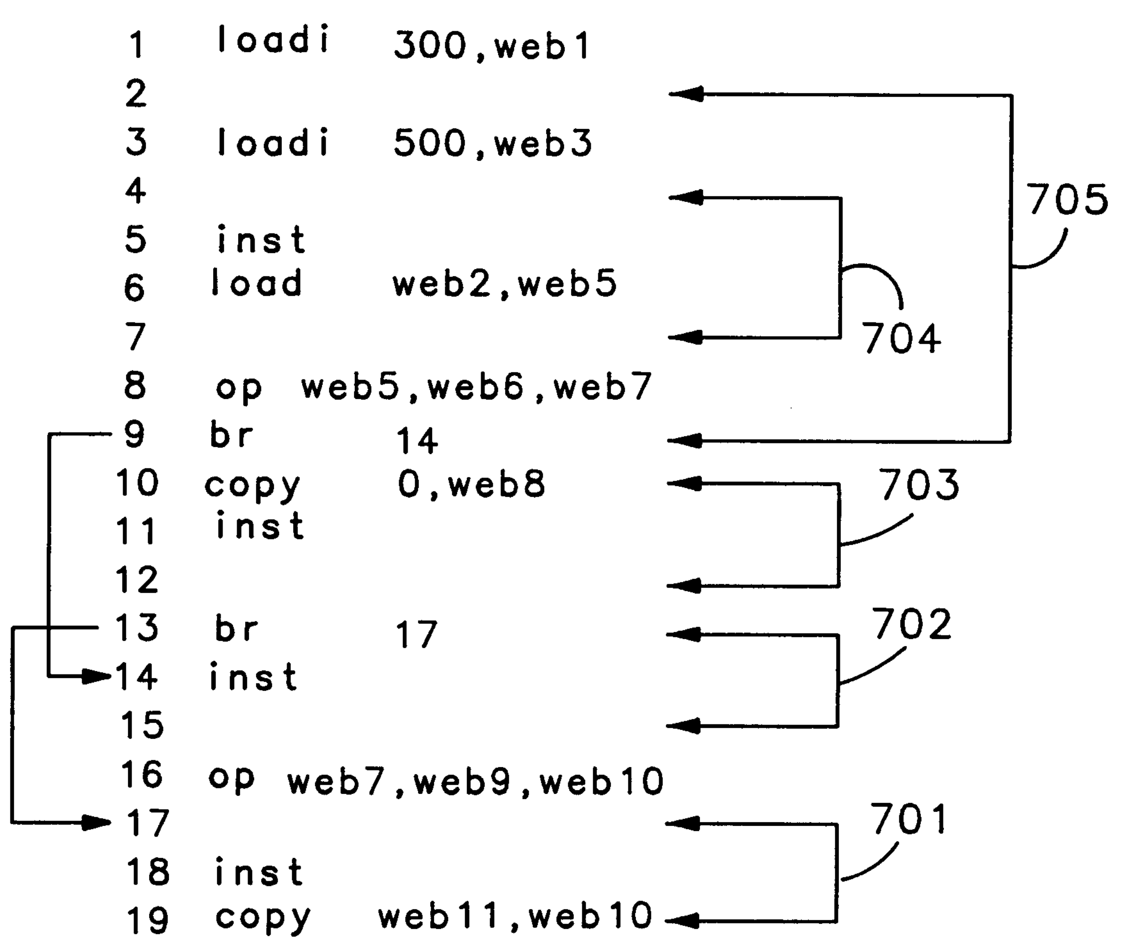
```
1   loadi  300,web1
2   store  web,web2
3   loadi  500,web3
4   store  web3,web4
5   inst
6   load   web2,web5
7   load   web4,web5
8   op  web5,web6,web7
9   br     14
10  store  0,web8
11  inst
12  load   web8,web7
13  br     17
14  inst
15  load   web2,web9
16  op  web7,web9,web10
17  store  web7,web11
18  inst
19  load   web11,web10
```

FIG 7



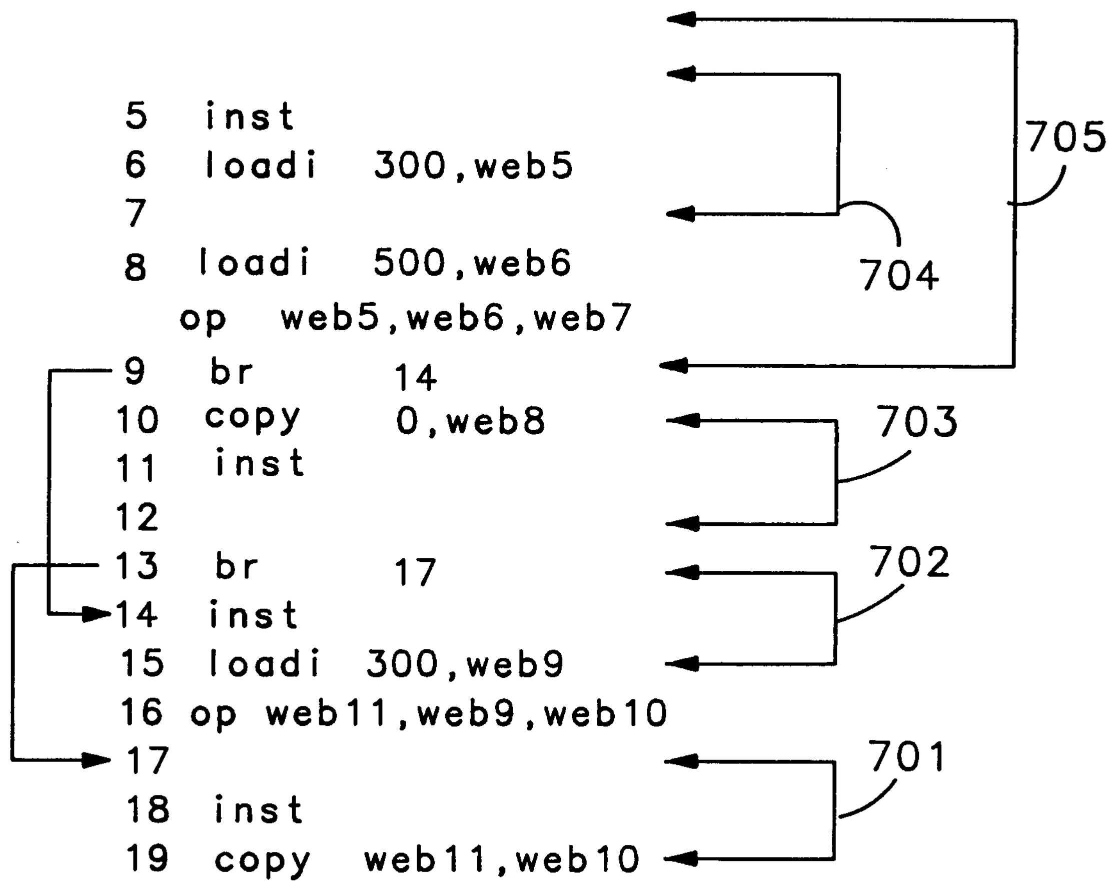
x
y
z
131
131
131
3
10
**
register
14
16
**
register
18
20
**
register
11
13
**
register
5
8
**
register
705
702
701
703
704
132
133
134
135

FIG 7A



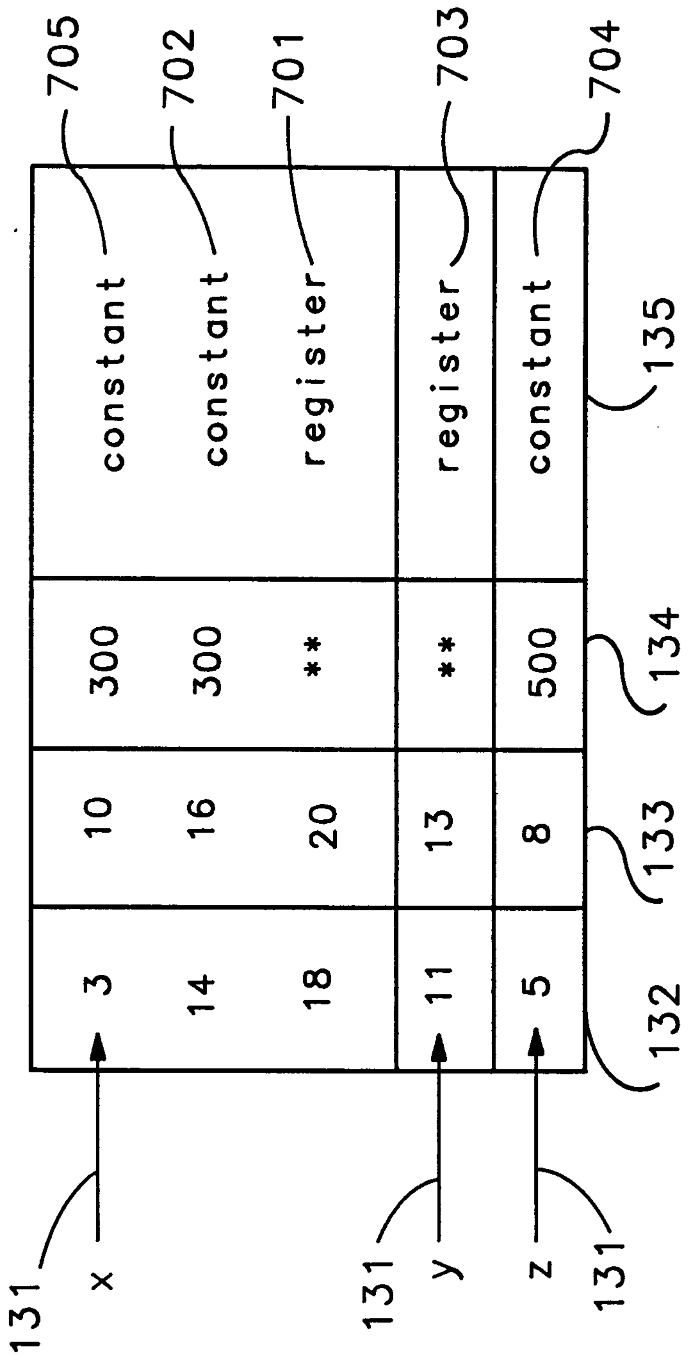
```
1   loadi   300,web1
2
3   loadi   500,web3
4
5   inst
6   load    web2,web5
7
8   op  web5,web6,web7
9   br      14
10  copy    0,web8
11  inst
12
13  br      17
14  inst
15
16  op  web7,web9,web10
17
18  inst
19  copy   web11,web10
```

FIG 8



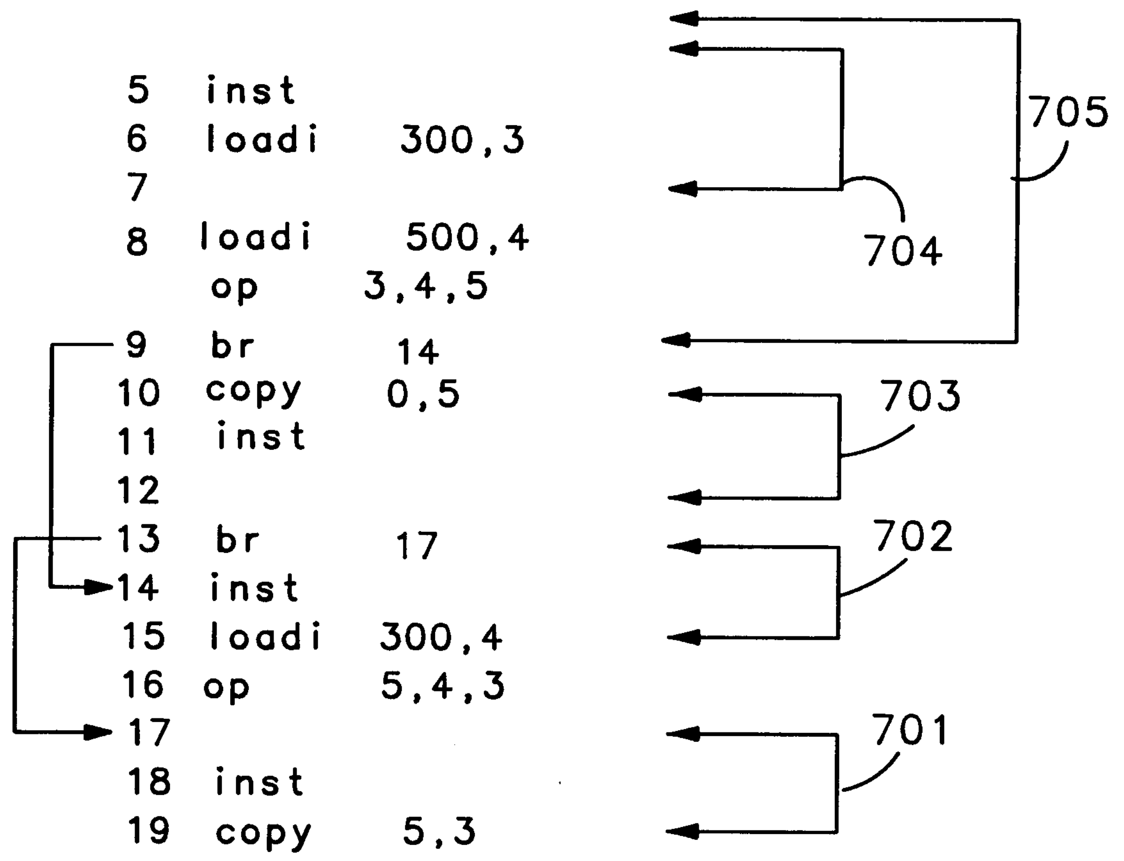
```
5   inst
6   loadi   300,web5
7
8   loadi   500,web6
    op   web5,web6,web7
9   br       14
10  copy     0,web8
11  inst
12
13  br       17
14  inst
15  loadi  300,web9
16  op web11,web9,web10
17
18  inst
19  copy   web11,web10
```

FIG 9



x	3	10	300	constant
	14	16	300	constant
	18	20	**	register
y	11	13	**	register
z	5	8	500	constant

FIG 9A



```
5   inst
6   loadi    300,3
7
8   loadi    500,4
    op      3,4,5
9   br       14
10  copy     0,5
11  inst
12
13  br       17
14  inst
15  loadi    300,4
16  op       5,4,3
17
18  inst
19  copy     5,3
```

FIG 10

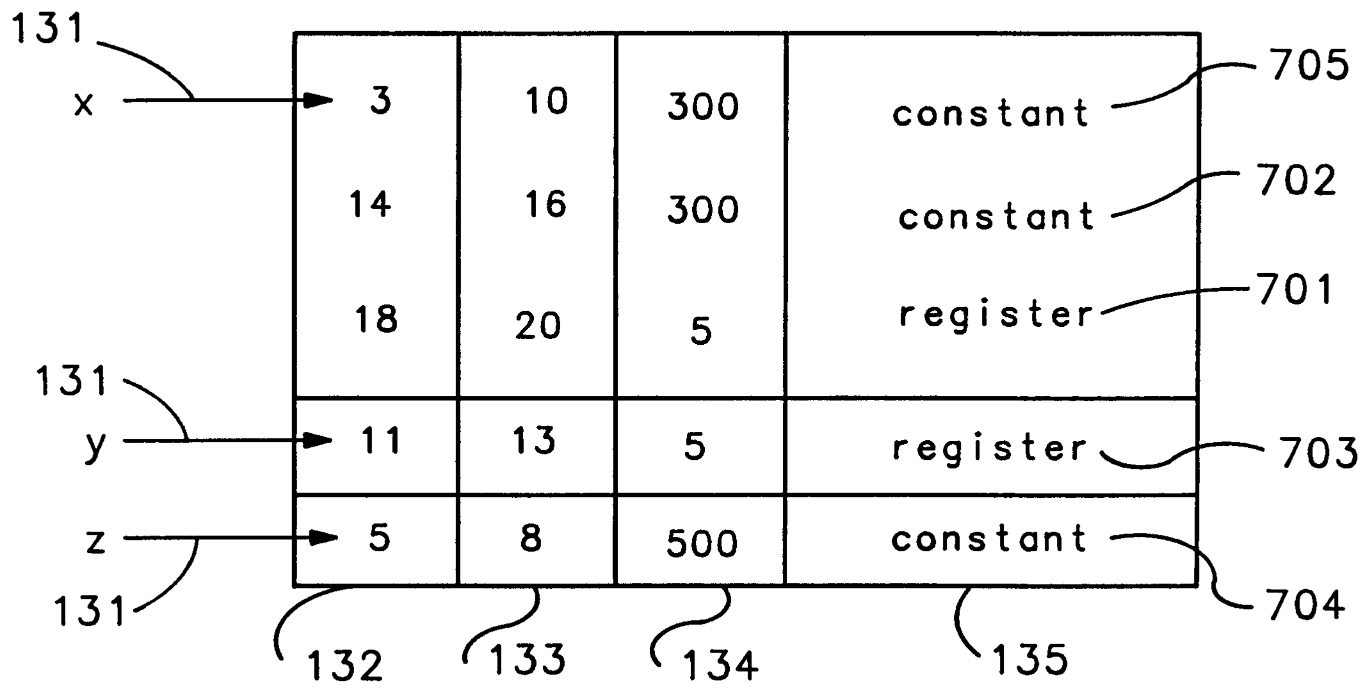
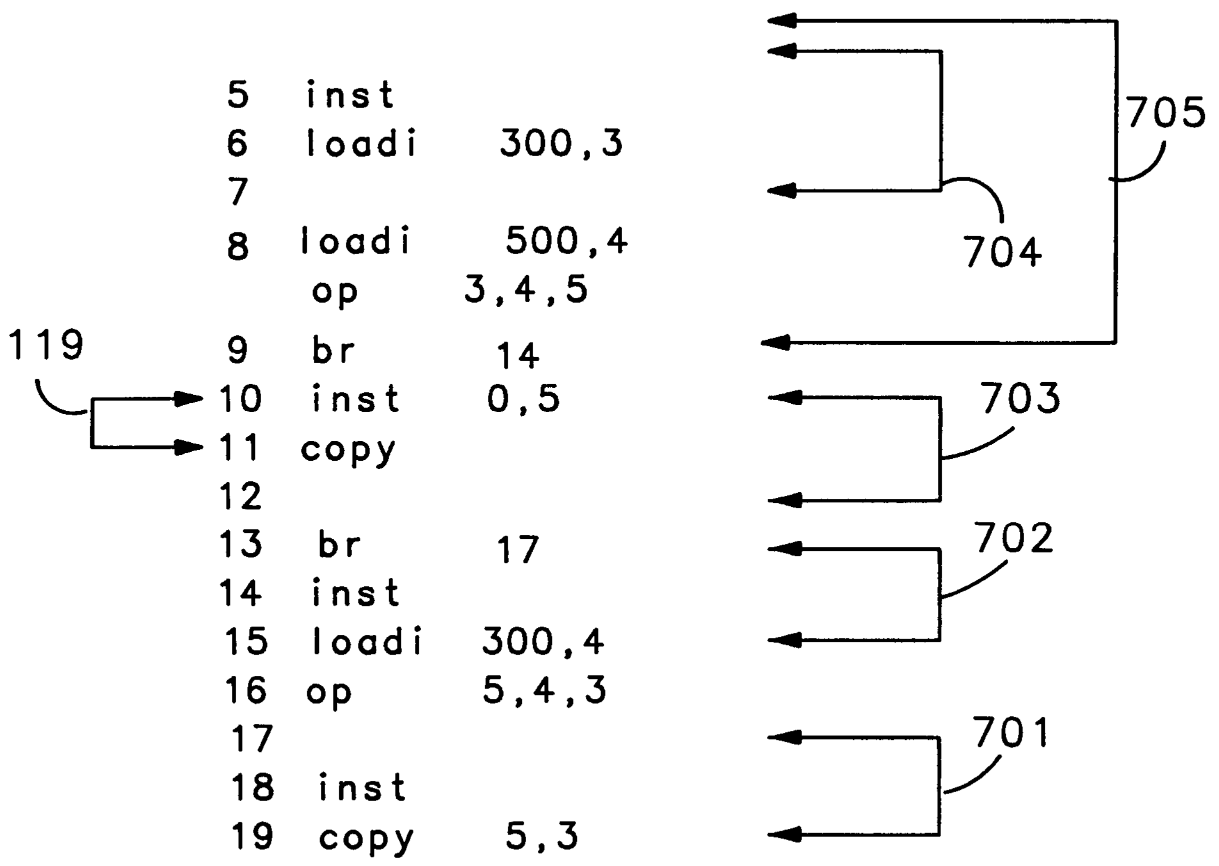

131
x
3
10
300
constant
705
14
16
300
constant
702
18
20
5
register
701
131
y
11
13
5
register
703
z
5
8
500
constant
704
131
132
133
134
135

FIG 10A



```
5   inst
6   loadi   300,3
7
8   loadi   500,4
    op      3,4,5
9   br      14
10  inst    0,5
11  copy
12
13  br      17
14  inst
15  loadi   300,4
16  op      5,4,3
17
18  inst
19  copy    5,3
```

FIG 11

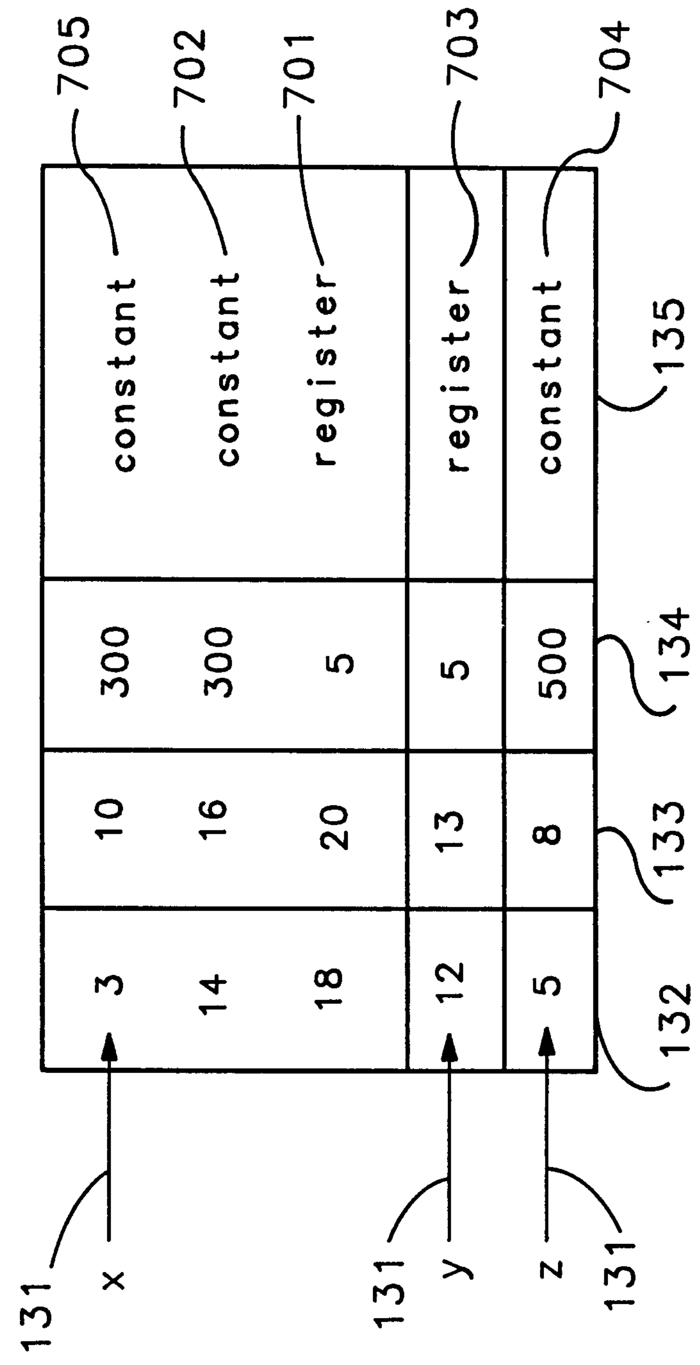

705
702
701
703
704
constant
constant
register
register
constant
300
300
5
5
500
10
16
20
13
8
3
14
18
12
5
135
134
133
132
131
x
131
y
z
131

FIG 11A